The ANML Language

David E. Smith **Jeremy Frank**
Intelligent Systems Division
NASA Ames Research Center
Moffet Field, CA 94035–1000
{*david.smith jeremy.d.frank*}*@nasa.gov*

William Cushing
Dept. of Computer Science and Engineering
Arizona State University
Tempe, AZ 85281
william.cushing@asu.edu

Abstract

The Action Notation Modeling Language (ANML) provides a high-level, convenient, and succinct alternative to existing planning languages such as PDDL, the IxTeT language, and languages developed at NASA, such as the EUROPA modeling language (NDDL), and the ASPEN modeling language (AML). ANML is based on strong notions of action and state (like PDDL, IxTeT, and AML), uses a variable/value model (like IxTeT, NDDL and AML), supports rich temporal constraints (like IxTeT, NDDL and AML), and provides simple, convenient idioms for expressing the most common forms of action conditions, effects, and resource usage. The language supports both generative and HTN planning models in a uniform framework and has a clear, well-defined semantics. Despite the richness of the language, it is possible to translate it into PDDL, with some increase in the number and complexity of operators and conditions. In this paper, we describe the central and unique features of the ANML language and sketch key parts of the translation to PDDL.

Introduction

The PDDL family of languages have become the standard for the academic planning community and continue to be developed and used for the semi-annual International Planning Competitions. Many example domains are available in PDDL, and many effective planners and techniques have been developed for different versions of the language. PDDL2.1 (Fox & Long 2003), and subsequent versions allow *durative* actions, and continuous numeric variables. This permits simple modeling of time, concurrency, and resources. However, the languages are still primarily proposition based, and do not directly support more complex temporal conditions, effects, or constraints. It is also cumbersome to express common patterns of change and resource usage, and it is easy to make errors in doing so as noted in (Cushing & Smith 2007). PDDL3.0 (Gerevini & Long 2005) has introduced the ability to model more complex temporal constraints and preferences on goals, but the capabilities do not extend to actions or conditions.

There are a number of different languages that have been developed and used at NASA for modeling planning domains and problems. Chief among these are the NDDL language (Bedrax-Weiss *et al.* 2005) developed for the EUROPA2 planner at Ames Research Center, and the ASPEN Modeling Language (AML) (Sherwood *et al.* 2005) developed at JPL. While both of these languages have their strong points, they are quite different in their basic concepts and assumptions, and both have significant weaknesses. The NDDL language facilitates the description of constraints among intervals. It characterizes every state or activity as an interval and specifies the possible temporal relationships between those intervals. However, NDDL has no notion of action, state, fact, or goal. A planning problem in NDDL simply involves filling out the set of timelines with intervals so that there are no gaps and all the constraints are obeyed. NDDL models are often quite verbose, unintuitive, and contain redundant constraints. It is also easy to make modeling mistakes in the language, and debugging models is difficult. In contrast, AML is based on fairly intuitive notions of action and state, and contains many convenient constructs for describing resource usage and temporal constraints. However, the language is fundamentally geared towards HTN planning, and does not fully support generative planning.

```
action Navigate (location from, to) {
  duration := 5 ;
  [all] { arm == stowed ;
          position == from :-> to ;
          batterycharge :consumes 2.0 } }
```

Figure 1: A simple ANML action.

The Action Notation Modeling Language (ANML) is being developed in an effort to: 1) provide a high level, convenient, and succinct alternative to existing languages, 2) support both generative and HTN planning models in a uniform framework, 3) provide a language with clear, well-defined semantics, and 4) allow greater compatibility with the evolving PDDL family of languages. ANML is based on strong notions of action and state (like AML and PDDL), uses a variable/value representation (like NDDL and AML), supports rich temporal constraints (like NDDL and AML), and provides simple, convenient idioms for expressing the most common forms of action conditions and effects. As an example action description in ANML, we can express a simple high-level navigate action for a rover as shown in Figure 1.

Navigate has two location parameters from and to, and (for simplicity) a fixed duration of 5. The *temporal qualifier* [all] states that the three following statements apply to the entire duration of the action. The first is a condition stating that

```
(:durative-action navigate
    :parameters (?from - location ?to - location)
    :duration (= ?duration 5)
    :condition (and (at start (position ?from))
                    (at start (stowed))
                    (over all (stowed))
                    (at start (>= (batterycharge) 2.0)))
    :effect (and (at start (decrease (batterycharge) 2.0))
                 (at start (not (position ?from)))
                 (at end (position ?to))))
```

Figure 2: The equivalent PDDL2.1 action.

the arm must remain stowed over the entire action. The second is a combination of a condition and two effects stating that the location must initially be the location from, is undefined in the interim, and will be the location to at the end of the action. The third is an effect stating that the action consumes two units of energy. Among other things, more complex temporal qualifiers are possible and more complex functional expressions are possible for duration and energy consumption.

For comparison purposes, Figure 2 shows an equivalent model in PDDL2.1. There are a number of syntactic differences between the two descriptions, but these are relatively unimportant. The more significant differences in both size and simplicity are due partly to ANML's variable/value representation, but also to ANML's more powerful constructs for describing change. In the ANML description, we have not partitioned the statements into conditions and effects. Instead, we have described what happens to each relevant variable in turn.

In the sections that follow, we explain these capabilities in greater detail, and highlight the novel features of the ANML language. We focus on the powerful and concise constructs in ANML for temporal qualification, for describing change over the course of an action, for describing resource usage, and for integrating task decomposition with traditional action models. To do this, we must first introduce the basic entities of the ANML language, time varying *variables* and *functions*.

Basic Declarations

Variables, and functions in ANML are typed. There are a number of built-in types, including: int, float, bool, string, object, and vector. It is also possible to declare new types in ANML. For example:

```
type positiveFloat := float [0.0, inff] ;
type rover := {spirit, opportunity, pathfinder} ;
type color := {blue, green, red, yellow, purple} ;
```

defines the type positiveFloat as a specialization of the type float, and defines the types rover and color by enumerating all members of the type. It is also possible to define specialized vector types, e.g:

```
type location < vector( positiveInt x, y ) ;
type path < vector( location from, to ) ;
```

which defines locations to be a subset ($<$) of vectors of two integer elements, x and y, and paths to be a subset of vectors of two locations, from and to. ANML also allows the definition of more complex structured types, but we will not discuss this here.

Variables

When declaring a variable in ANML, one must specify the domain of the variable. The declaration therefore consists of the keyword variable followed by a type, followed by the variable name. Any predefined or user-defined type can be used for this purpose. For example:

```
variable string sampleName, pictureName ;
variable float [0.0, inff] batterycharge, wheelCurrent ;
variable color {blue, red, green} filterColor ;
variable location position ;
```

are all legitimate variable declarations.

For convenience, variables can be initialized in a declaration. For example:

```
variable float roverSpeed := 30.0 ;
```

is equivalent to stating:

```
variable float roverSpeed ;
[start] roverSpeed := 30 ;
```

Vectors can also be initialized in this way. For example:

```
variable vector( int x, y ) position := (5, 20) ;
```

is equivalent to stating:

```
variable vector( int x, y ) position ;
[start] { position.x := 5 ;
          position.y := 20 } ;
```

Functions

Functions in ANML are essentially parameterized variables. So when declaring a function symbol, in addition to specifying the range of the function, one must also specify the domains of the parameters or arguments. The declaration for a function symbol therefore consists of the keyword function followed by a type, followed by a function symbol and its typed argument list. For example:

```
function float [0, inff] batterycharge(rover r) ;
```

indicates that the function batterycharge of a rover is a positive float. As with variables, functions in ANML are implicit functions of time. In fact, variables in ANML can be considered as functions having zero arguments.

Actions

An ANML action description consists of: the action name, a typed parameter list, a duration assignment (optional), local variable or function declarations (optional), and one or more temporally qualified conditions, effects, or change statements. Consider the example ANML action shown in Figure 1. The action name and typed parameter list are shown in the first line, and the duration assignment is shown in the second line. The action does not contain any local variable definitions, but we could have done something like define a local variable for energy use in terms of duration, and then express the consumption using this local variable:

```
variable float energy := duration * use-rate ;
[all] { batterycharge :consumes energy } ;
```

The main body of our example action consists of the three lines:

```
  [all] { arm == stowed ;
          position == from :-> to ;
          batterycharge :consumes 2.0 } ;
```

In general, the body can express simple conditions (like the first line above), simple effects, or more complex combinations of conditions and effects (lines 2 and 3). To be more precise, these statements take the form:

$$\textit{Temporal Qualifier}\ \{\phi_1; \ldots; \phi_n\}$$

where *Temporal Qualifier* is something like [start] or [all], and each of the ϕ_i is a condition, effect, or change expression. A simple condition expression is of the form:

variable relation expression

where *relation* is typically ==, in, or a numeric comparator ($<$, $\leq$, $>$, $\geq$). The expression is frequently a constant, or another variable, but can be a set, interval or an algebraic expression of numeric variables and constants.

A simple effect expression is of the form:

variable assignment expression

where *assignment* is one of := or :in. The expression is the same as for conditions, but can also be undefined (one is not allowed to condition on a variable being undefined). For convenience, we allow assignment to undefined to be abbreviated with with :_ e.g:

```
position :_ ;
```

Conditions and effects can be distinguished because the relations in conditions are distinct from the assignment operators used in effects. Because of this, there is no need to separate them or delineate them with a keyword as in PDDL.

Combinations of conditions and effects only make sense over intervals. Typically, for closed intervals, they are of the form:

```
variable   relation     expression1
           assignment   expression2
           assignment   expression3 ;
```

The relation and expression in the first line expresses a condition that must hold at the beginning of the interval. For example, in the expression:

```
[all] { position == from :_ := to } ;
```

the condition is position == from, which must hold at the beginning of the interval. The assignment :_ is an effect indicating the value of the variable over the interior of the interval, in this case undefined. The last assignment and expression is also an effect indicating the final value of the variable at the end of the interval, in this case the value to. Thus the above expression could be expressed as a simple condition and two simple effects:

```
[start] position == from ;
(all) position :_ ;
[end] position := to ;
```

This particular form, with assignment to undefined over the interim and assignment to a final value at the end, is so common that we allow the further shorthand :-> (goes to) for the combination :_ :=. Thus the above composite statement with condition and two effects can be written as:

```
[all] { position == from :-> to } ;
```

Temporal Qualifiers

A temporal qualification indicates the time or time period over which a variable has a particular value or changes its value. The simplest temporal qualifier $[t]\ \phi$ specifies a single time point t at which a condition or effect ϕ holds. Time points are specified relative to start and end, meaning the start and end of the action respectively, e.g:

```
[start] position == from ;
[end-5] heater := on ;
```

Intervals are also permitted as temporal qualifiers, indicating that the condition holds, or the assignment is enforced over the entire interval, e.g:

```
[start+5, end-2) heater == on ;
```

Intervals can be open or closed on either end. In addition, all is a shorthand for the common interval from start to end (but one must still specify whether it is closed or open at either end).

A third useful temporal qualifier is of the form:

i contains $d\ \phi$

meaning that the condition phi holds for at least duration d within interval i. For example:

```
[all] contains [3] heater == on ;
```

specifies that the the heater must be on for at least a closed interval of duration 3 within the interval all. This quantifier is existential in nature (there exists an interval such that ...) and can therefore only be used with conditions.

If the duration is omitted for contains the entire interval for the condition must be contained. For example:

```
[all] contains heater == on ;
```

specifies that there must be a (maximal) interval with the heater on, contained entirely within the interval [all]. This will turn out to be particularly useful for referring to actions.

Relative Change

For numeric variables, it is often useful to specify effects relative to the existing value, rather than in absolute terms. For example we might want to specify that a particular drilling operation advances the drill a certain amount beyond its current depth. We could state this as:

```
[end] depth := depth + increment ;
```

Some languages make this a bit easier by allowing additional operators like += and −=. In ANML we do this by specifying change on the "delta" of the depth variable rather than on the depth variable itself. For the above example we would say:

```
[end] △depth := increment ;
```

The meaning of this is that the depth variable changes by the amount increment.[1] The reason we take this approach is that it allows us to conveniently say other more difficult things like:

[1]The delta symbol (△) is a bit hard to type on most keyboards, so we use the carat symbol (^) as a substitute.

```
(all) △depth :-> increment ;
(all) △depth :in [0, increment] := increment ;
```

The first of these implies that the change in depth is undefined over the course of the interval, before taking on the final value increment. The second implies that the change in depth is bounded by the interval [0, increment] over the course of the interval, before taking on the final value increment. These statements are considerably more cumbersome using only the basic primitives, because they require defining a temporary variable to hold the initial value of the depth variable, e.g:

```
variable float start-depth := depth ;
(all) depth :_ ;
[end] depth := start-depth + increment ;
```

As we will see in the next section, composite incremental change statements will prove extremely convenient for describing resource consumption and production.

Resources

Resources are common and convenient abstractions in many planning and scheduling applications. There are many different kinds of resources – they can be discrete or continuous, and consumable or reusable.[2] Discrete resources can also be *unit-capacity*, or *multi-capacity*.

As far as ANML is concerned, resources are just numeric variables. They are therefore declared in the same way:

```
variable float [10.0, 100.0] batterycharge := 50.0 ;
variable integer [0, 12] sample-bags := 12 ;
```

It is the usage of these resources where ANML provides additional facilities beyond the notation described so far. We start with *reusable* resources.

Usage

A *reusable* resource is one that is *consumed* at the beginning of an action, but given back (or *produced*) at the end. Using the mechanisms for relative change that we introduced above, we could express this as:

```
[start] △resource := –quantity ;
[end] △resource := quantity ;
```

However, because resource use is so common, we have introduced a more convenient way of expressing this pair of effects:

```
[all] resource :uses quantity ;
```

It is important to note that for a resource effect like this, the condition:

```
[start] resource >= quantity ;
```

is implicit because of the definition of the resource variable. For example, a declaration of batterycharge as:

```
variable float [10.0, 100.0] batterycharge ;
```

implicitly requires that the quantity remain within the interval [10.0, 100.0]. Any action that would violate these bounds would not be legal.

[2]The first of these dimensions, *discrete or continuous*, is a property of the resource variable itself. The second, *consumable or reusable* is a property of how the resource is used. In fact, it is entirely possible for a quantity to be a reusable resource for one action and a consumable resource for another.

Consumption and Production

As we hinted above, resource usage can be thought of as a pair of consumption and production effects. Using the mechanisms for modeling relative change, we can model instantaneous resource consumption and production as:

```
[t] △resource := –quantity ;
[t] △resource := quantity ;
```

For convenience and clarity we allow these to be stated as:

```
[t] resource :consumes quantity ;
[t] resource :produces quantity ;
```

When dealing with consumption and production of resources, it is common to make a conservative modeling assumption that consumption occurs at the beginning of an action and production occurs at the end. In reality, consumption and production usually occur gradually over the course of actions, although the actual function may be complex or even unknown. The conservative discretization ensures that there is enough of a resource at the beginning of a consumption action, and that we do not rely on any production actions until their end. In effect, this approach tracks a lower bound for the resource, and guarantees that plans will never violate the lower bound limit for the resource variable. While this works well in many situations, it can run into trouble if the resource variable also has an upper bound, or *capacity*, such as for a battery, fuel tank, or storage container (Cushing & Smith 2007). Figure 3 illustrates the problem. A navigate action is performed while the battery is being recharged (by solar panels). The conservative lower bound envelope for the two actions remains within the allowed range for the battery. However, since production actually occurs before the end of the recharge action, the actual charge envelope will exceed battery capacity. This kind of problem can occur any time there is the possibility of simultaneous consumption and production, and the resource has a capacity limit. Cushing and Smith (Cushing & Smith 2007) discuss this problem in detail, and discuss some alternative approaches to dealing with it. The most obvious way of dealing with this problem (without resorting to detailed modeling and reasoning about continuous consumption and production) is to keep track of both a lower bound and an upper bound for a resource. For

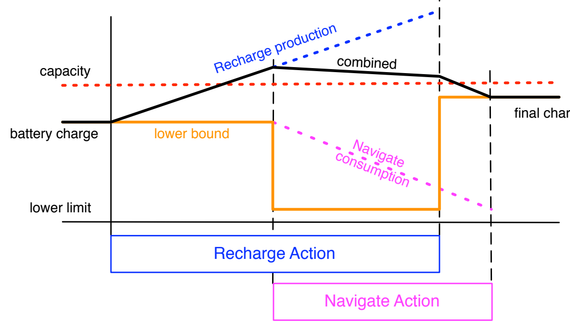

Figure 3: Illustration of simultaneous consumption and production activities. Although the lower bound remains within the allowed range for the resource, the actual value exceeds resource capacity unless the consumption activity is started earlier.

example, we could model the behavior of a consumption action as:

[start] $\triangle$resource_lb := –quantity ;
[end] $\triangle$resource_ub := –quantity ;

and a production action as:

[start] $\triangle$resource_ub := quantity ;
[end] $\triangle$resource_lb := quantity ;

While sound, this requires that we define and keep track of two explicit variables for each resource, and get the timing and polarity of the conditions and effects right. We regard this as both complex and cumbersome, and likely to result in many modeling errors. Instead, we think that the user should be able to simply state that consumption or production occurs over the course of an action (or interval). The planner should manage the details of keeping track of lower and upper bounds for variables in order to assure soundness. Using the notation for relative change, and for compound change statements, we would express a typical resource consumption as:

(all] $\triangle$resource :in [–quantity,0] := –quantity ;

This states that over the course of the action, the resource change is guaranteed to be between zero (upper bound) and –quantity (lower bound) and will be at the lower bound at the end of the interval. We define the shorthand:

[all] resource :consumes quantity ;

to mean precisely this for an interval; the quantity change is bounded over the course of the interval, and ends with the specified change. Similarly, resource production over an interval:

[all] resource :produces quantity ;

is defined as:

(all] $\triangle$resource :in [0,quantity] := quantity ;

The :consumes and :produces statements are extraordinarily convenient and powerful, because they allow the user to specify complex production and consumption activities without requiring details of the actual consumption and production functions, and without requiring that the user explicitly provide upper and lower bound discrete approximations.

HTN Decomposition

Hierarchical Task Networks (HTNs) and task decomposition planning techniques are used for many practical planning systems and applications (Wilkins 1988; Chien *et al.* 2000; Nau *et al.* 2003; Castillo *et al.* 2006). For HTN systems, goals are stated in terms of high level *tasks* to be performed, and *methods* allow for the decomposition of these tasks into lower level tasks and primitive actions. Many within the planning community have argued against the HTN representation and task decomposition on the grounds that they describe what actions *should* be used for (their purpose), rather than what actions *do* (their conditions and effects). As such, this representation can lead to systems that are highly tailored to a certain class of problems, but can be brittle if they are asked to solve problems that require using actions in unanticipated ways. All of this is true. Nevertheless, there are some good arguments for allowing task decomposition in a modeling language:

1. Some planning domains seem to be more naturally expressed in terms of task decomposition.
2. There are situations where the modeler may be able to characterize ways of achieving tasks without having a clear understanding of the effects and conditions of the underlying actions.

In developing ANML, we did not wish to unduly constrain the modeler, or force the modeler to describe actions at an unnecessarily fine level of detail. As a result, ANML allows the specification and use of action decomposition by allowing decomposition expressions within action descriptions. We have done this in a novel way that allows action decomposition to be fully integrated with ordinary action descriptions containing conditions and effects. There are three things necessary to make this work:

1. Each action (instance) A is regarded as having an implicit effect proposition of the same name (A) over its execution. In other words:

   [all] A == false := true := false ;

2. We allow these "action" propositions to be used as ordinary conditions within action descriptions.
3. We allow relative ordering constraints on conditions.

Allowing relative ordering constraints among conditions requires some additional machinery which we now describe.

Relative Ordering Constraints

Suppose that we have two action conditions p and q, that must be true at some point during an action, e.g:

[all] contains { p ; q } ;

Now imagine that we also require that p become true before q. Using the machinery presented so far, we could specify explicit intervals for p and q that guaranteed this, but we could not simply specify a relative ordering constraint between the two conditions. To do this, we generalize the keywords start and end to allow them to refer to the start and end of specific actions, e.g: start(p).[3] We can then specify the desired relative ordering constraint as:

start(p) < start(q) ;

The entire action condition would therefore be:

[all] contains { p ; q } ;
start(p) < start(q) ;

This ordering capability is quite general and allows us to specify complex temporal constraints among arbitrary sets of conditions. For convenience, we introduce the shorthand ordered($p_1, \ldots, p_k$) to refer to an ordered set of conditions. In other words:

[all] contains ordered($p_1, \ldots, p_k$) ;

is equivalent to naming each of the conditions, and specifying temporal constraints between each successive pair:

[3]More generally, we need to be able to name specific instances of actions and conditions to be able to distinguish between multiple instances. This can be done using the construct n : p.

```
[all] contains { n1 : p1 ; ... ; nk : pk } ;
end(n1) ≤ start(n2) ;
...
end(nk−1) ≤ start(nk) ;
```

A similar shorthand unordered(p_1, ..., p_k) can be used to refer to an unordered set of conditions. Together, ordered and unordered can be used to describe a partial ordering of conditions. For example:

```
ordered( p, unordered( q, ordered( r, s )), t ) ;
```

corresponds to the partial order shown in Figure 4.

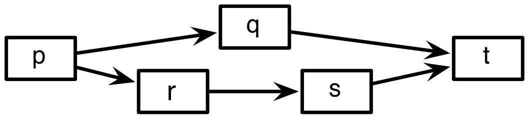

Figure 4: A partially ordered set of conditions.

Decompositions

With the ability to express relative ordering constraints on conditions, we now have the tools necessary to express decomposition in an action description. As an example, suppose we want to specify a method for the high level task of CollectSample, which can be decomposed into four primitive actions of unstowing the instrument, placing the instrument, taking the sample, and stowing the instrument. We can express this in ANML as:

```
action CollectSample(location l) {
  [all] position == l ;
  [all] contains ordered( Unstow, Place(l),
                          TakeSample(l), Stow ) ;
  ...} ;
```

In effect, this says that in order to successfully perform the CollectSample action, we must successfully perform the Unstow, Place( ), TakeSample( ), and Stow actions, in order.

Translation to PDDL

Surprisingly enough, in most cases, ANML can be translated into PDDL 2.2, with only a modest increase in the number of actions, although the resulting translation is not particularly clear or easy to work with.

Theorem 1 *If all actions in an ANML model are self-mutex (no two identical instances can execute simultaneously)*[4] *then the model can be translated into PDDL2.2. The PDDL 2.2 translation will generally require additional actions, but the number of additional actions can be bounded as follows:*

- *each condition with a* contains *temporal qualification and a duration requires one additional action*
- *each condition with a* contains *temporal qualification but no duration requires at most three additional actions*
- *if the conditions and effects of an action refer to k intermediate timepoints (between start and end) k+1 additional actions are required.*
- *each relative ordering constraint may require upto three additional actions, but in most cases no additional actions are required.*
- *one additional action is required for each subtask in a decomposition, and additional actions may be required for the relative ordering constraints as specified above.*

[4]Actually this is much stronger than necessary – only certain complex actions (actions with intermediate conditions or effects, decompositions, relative ordering, etc.) need to be self-mutex for the construction to work. Even this restriction can be relaxed if the number of copies of a complex action that can be executed simultaneously is bounded.

Showing the translation for every ANML construct requires considerably more space than is available here. We therefore only sketch the argument by showing the translation for some of the more interesting cases.

Proof sketch: Translating the variable/value representation in ANML to a propositional representation is straightforward – for each discrete variable v we introduce a predicate $V(y)$ of one argument such that $V(a)$ is true just in case $v = a$. Similarly for functions in ANML. For each assignment effect in ANML there will be a corresponding delete effect for the old value and an add effect for the new value in PDDL. Composite statements in ANML are broken down into the condition and effects according to the definition given previously.

Numeric variables in ANML are translated directly to numeric variables in PDDL. Resource effects in ANML can be broken down into their component effects as described previously. The only tricky part is that consumption and production over an interval (e.g. [all] {v :consumes a}) requires that we define explicit upper and lower bound variables in PDDL. Additional conditions must also be added to the PDDL actions to ensure that variable values remain within their defined ranges.

The more interesting and difficult parts of the translation involve dealing with the ANML temporal qualifier contains, with intermediate conditions and effects, and with relative ordering constraints among conditions.

Figure 5 shows how to construct a PDDL translation of an ANML action A having a condition of the form:

```
[all] contains dur [d] {p} ;
```

Following the notation of (Halsey, Long, & Fox 2004; Cushing *et al.* 2007) actions are shown in boxes with durations in square brackets. Propositions above an action are start, over all, and end conditions respectively, and propositions below an action are start and end effects respectively. The construction in Figure 5 works by introducing an auxiliary action P of duration d that is: 1) forced to occur within A, and 2) requires that the proposition p be true over its entire duration.

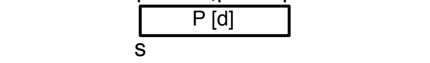

Figure 5: Establishing [all] contains dur [d] {p} in PDDL.

Figure 6 shows a similar construction for establishing conditions of the form

[all] contains B ;

where B is an action. In this case the auxiliary action B* is forced to occur within A, and forces B to be concurrent.

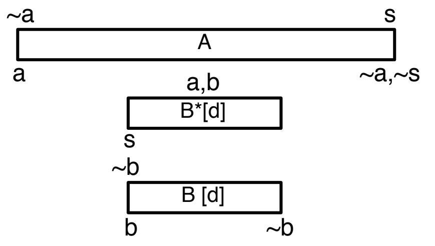

Figure 6: Establishing [all] contains B in PDDL.

Translating actions with intermediate conditions and effects requires introducing auxiliary actions that break up the action A into a sequence of ordered pieces. Figure 7 shows how this can be done for an action with two intermediate effects:

[start+t1] e1 ;
[start+t2] e2 ;

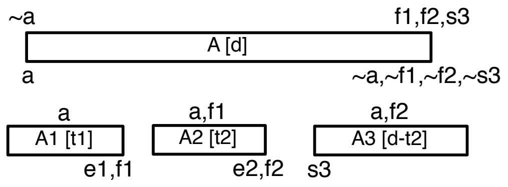

Figure 7: Establishing two intermediate effects e1 and e2.

This can be easily generalized to handle any sequence of intermediate point conditions and effects, as well as interval conditions.[5]

Finally, we consider action decompositions such as that shown in Figure 4. We have shown previously (Figure 6) how to force the subtasks in a decomposition to occur within the parent task. The only thing remaining is to enforce partial ordering constraints among the tasks. Given two subtasks A and B this is generally quite easy. Figure 8 shows how this can be done for two of the common cases. The only difficult case, where an auxiliary action is required, is shown in Figure 9.

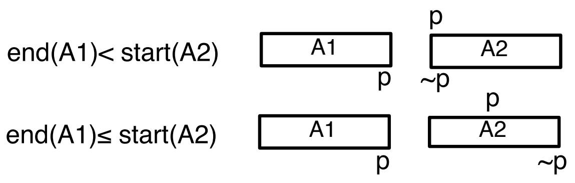

Figure 8: Enforcing ordering between two actions.

Orderings with numeric constants, eg: end(A1) + 5 ≤ start(A2) can be handled by introducing auxiliary actions in a manner similar to the construction shown in Figure 9. Orderings between propositional conditions can also be handled with similar constructions. ■

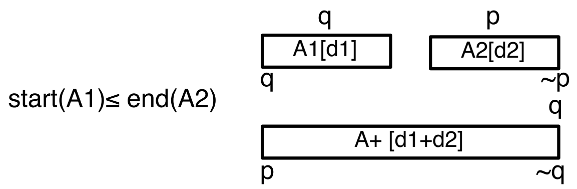

Figure 9: The most difficult ordering constraint.

Related Work

The ANML language has taken inspiration from several existing languages. Among these, the NDDL language has the most powerful capability for expressing temporal constraints. However, these capabilities are not necessarily easy to understand or use, and we find many aspects of the language to be both cumbersome and difficult to use effectively. The AML language has very convenient and natural syntax for expressing resource usage, but the HTN nature of the language is limiting. The PDDL family of languages has been carefully developed, and is widely adopted in the research community. However, the propositional nature of the language, the limited constructs for describing change and resource usage, and the limited ability to model time and temporal constraints make it difficult or impossible to use for serious applications. The change notation in ANML most closely resembles constructs developed in the SAS family of languages (Jonsson & Bäckström 1994). However, in overall capability and style, the IxTeT language (Ghallab & Laruelle 1994) is perhaps the closest.

There have been a few previous attempts to merge HTN decomposition into more traditional action languages like PDDL (McDermott 1998; Castillo *et al.* 2006). However, these attempts have not been widely used or adopted because the two paradigms have had separate semantics and have not really been integrated. The semantics we ascribe to decompositions in ANML is quite different. Essentially, we regard a decomposition as simply being another set of conditions necessary for performing the action. In other words, if the subtasks can be performed in the order indicated, then the high-level task can be performed. This makes sense for several reasons:

1. If the decompositions for an action prove to be impossible (cannot be performed), then the action itself is not possible (or we do not know its outcome).
2. Multiple decompositions correspond to a disjunction of sets of conditions, and any one of these sets would be sufficient to accomplish the action.
3. Logical conditions can be mixed with decompositions.
4. It is consistent with, and can be seen as a generalization of allowing general temporal constraints among conditions.

Finally, some of the constructions developed for translating ANML into PDDL were inspired by those found in (Fox, Long, & Halsey 2004) and (Hoffmann *et al.* 2006).

[5]For all practical purposes, effects over an interval can be translated into a point effect at the beginning of the interval and a condition over the remainder of the interval guaranteeing that the effect still holds.

Conclusions and Future Work

In this paper, we have only sketched some of the key features of the evolving ANML language. In particular, we have described the powerful and concise constructs in ANML for temporal qualification, for describing change over the course of an action, for describing resource usage, and for integrating task decomposition with traditional action conditions and effects. There are additional features and details of the language that we have not mentioned, or have only glossed over. Among other things, there is the ability to express quantification, disjunctive conditions, conditional effects, and complex goals.

The language has evolved considerably since it was first conceived in 2006, largely due to efforts to model some International Space Station problems involving crew scheduling and procedures. Many of the differences may appear to be minor syntactic changes, but were actually motivated by a need to generalize a construct, or clarify semantics in ways we had not previously considered. A manual describing the language is available, along with an ANTLR grammar. We are in the process of building a translator from ANML into PDDL using the constructions outlined in this paper. We are also working on a translator from PDDL into ANML using techniques outlined in (Bernardini & Smith 2008). A translator from ANML into NDDL was developed previously, but currently only handles a subset of the language.

There is an additional effort underway to extend the syntax of ANML to allow description of continuous change, processes and exogenous events. We would like to be able to express resource usage as a function of time where known and convenient. However, our view here is not that a planner must necessarily be able to reason about continuous change. Instead, our view is that we should allow the user to naturally express the domain, at whatever level of detail is appropriate. It is then perfectly reasonable for a planner to make sound but incomplete approximations to effectively reason about the domain. As a case in point, reasoning about lower and upper bound envelopes is a useful approximation that is both computationally tractable, and perfectly adequate for many domains. However, we want the planner to be able to choose the approximation, rather than forcing the user to encode it explicitly. Thus in the case of continuous change, we would like to see the planner choose an appropriate discretization or linearization in order to enable sound, but effective, reasoning about the domain.

Acknowledgements

Thanks to Tony Barrett, Mark Boddy, Matthew Boyce, Maria Fox, Ari Jonsson, and Conor McGann for discussion and feedback on the language and its design. This work was supported by the Automation for Operations project of the NASA Exploration Technology Development Program.

References

Bedrax-Weiss, T.; McGann, C.; Bachmann, A.; Edgington, W.; and Iatauro, M. 2005. EUROPA2: User and contributor guide. Technical report, NASA Ames Research Center.

Bernardini, S., and Smith, D. 2008. Translating PDDL2.2. into a constraint-based variable/value language. In *ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling*.

Castillo, L.; Fdez-Olivares, J.; Garcia-Pérez, O.; and Palao, F. 2006. Efficiently handling temporal knowledge in an HTN planner. In *Proc. of the Sixteenth Intl. Conf. on Automated Planning and Scheduling (ICAPS-06)*.

Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; B.Smith; Fisher, F.; Barret, T.; Stebbins, G.; and Tran, D. 2000. ASPEN - Automated planning and scheduling for space missions operations. In *International Conference on Space Operations (SpaceOps 2000)*.

Cushing, W., and Smith, D. 2007. The perils of discrete resource models. In *ICAPS-07 Workshop on International Planning Competition: Past, Present and Future*.

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. 2007. When is temporal planning really temporal? In *Proc. of the Twentieth Intl. Joint Conf. on Artificial Intelligence (IJCAI-07)*.

Fox, M., and Long, D. 2003. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Fox, M.; Long, D.; and Halsey, K. 2004. An investigation into the expressive power of PDDL2.1. In *Proc. of the Sixteenth European Conf. on Artificial Intelligence (ECAI'04)*.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3: The language of the Fifth International Planning Competition. Technical report, Department of Electronics for Automation, University of Brescia.

Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proc. of the Second Intl. Conf. on Artificial Intelligence Planning Systems (AIPS-94)*, 61–67. AAAI Press.

Halsey, K.; Long, D.; and Fox, M. 2004. CRIKEY - a planner looking at the integration of scheduling and planning. In *Proc. of the Fourteenth Intl. Conf. on Automated Planning and Scheduling (ICAPS-04)*, 46–52.

Hoffmann, J.; Edelkamp, S.; Thiebaux, S.; Englert, R.; Liporace, F.; and Trueg, S. 2006. Engineering benchmarks for planning: the domains used in the deterministic part of IPC-4. *Journal of Artificial Intelligence Research* 26:453–541.

Jonsson, P., and Bäckström, C. 1994. Tractable planning with state variables by exploiting structural restrictions. In *Proc. of the Twelfth National Conf. on Artificial Intelligence (AAAI-94)*.

McDermott, D. 1998. PDDL – the Planning Domain Definition Language: Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Nau, D.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, J.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Sherwood, R.; Engelhardt, B.; Rabideau, G.; Chien, S.; and Knight, R. 2005. ASPEN user's guide. Technical Report D-15482, JPL.

Wilkins, D. E. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann.